LR Parsing

Mattox Beckman
<beckman@iit.edu>

July 4, 2006

1 Introduction

Recursive descent parsers are easy to write, but they need an LL grammar to work. If the grammar is not LL, it can be transformed (usually!), but the transformation itself may make it difficult to use the results of a parse.

LR parsing techniques allow any context-free grammar to be used in the parse, but the technique is more complex.

1.1 Objectives

- Be able to explain how LR parsing is different than LL parsing.
- Know how to generate the Characteristic Finite State Automata for a grammar.
- Know what the item sets mean.
- Know how to create the LR Action and Goto tables
- Know what a Shift-reduce conflict is, and how to fix it
- Know what a reduce-reduce conflict is, and how to fix it
- Know how to use the parse tables to generate an "execution trace" of a parse.

2 Top-Down Parsing

An LR parser solves the same problem as an LL parser, but it uses a different approach. Both scan the input from left to right (this is what the first L indicates), and build a parse tree out of the input.

The LL parsers we covered earlier are called ***top-down*** parsers: they start with the start symbol of the grammar as the root of the tree, and then build the tree by filling in the branches, from left to right.

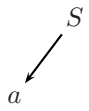
For a quick review, here is an example of LL parsing.

Suppose we have the following grammar:

$$\begin{array}{rcl} S & \rightarrow & a\ E\ z \\ E & \rightarrow & x\ y \end{array}$$

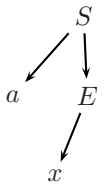
The language described by this grammar can only recognize one sentence: `a x y z`. Let's see what happens when it is parsed.

The LL parser will start with the symbol S. Upon reading `a`, it will select the appropriate rule[1], and will create a partial parse tree:



The second two branches have not been filled in yet. In reality, the first one hasn't either. The parse tree is being built by a recursive-descent parser. The function calls needed to create the S node are still being made: the first one has returned (the one that gets a), but the rest are still pending. But that is an implementation detail: the construction of the tree is the abstraction we are representing.

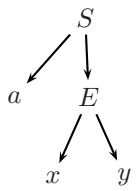
The parser will then start to build the middle branch. This will generate the E node. It will then read the `x` token. At this point, our tree looks like this:



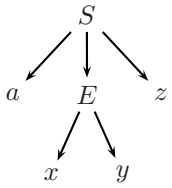
Notice how the tree is being filled in from the left side first. This is called a ***leftmost derivation***, and this is what the second L in LL represents.

[1] Actually, since there is only one rule, this could occur before the `a` is seen.

After the parser reads the next token, it has enough information to really build the E node of the parse tree. The function building that node returns the tree to the caller: the function building the S. Our tree now looks like this:



Finally, the parser reads the `z`, filling in the last branch of the S node, and returning the parse tree to whoever asked for it.



3 Bottom Up Parsing

In contrast to LL parsers, LR parsers work by reading the input one token at a time, pushing the tokens onto a stack. This operation is called ***shifting***. These tokens are the leaves of the parse tree. When enough leaves have been pushed onto the stack, the parser creates a tree branch representing the appropriate non-terminal symbol, pops the leaves off the stack and makes them children of the new branch, and then pushes the branch back onto the stack. This operation is called ***reduction***.

The LR parse is conducted by using a ***push-down automata***, a state machine that has a stack. The NFAs and DFAs we discussed during the lexical analysis did not have a stack, and this severely limited the kinds of things they could express. These state machines will be more powerful.

An automata for parsing LR grammars is called a ***Characteristic Finite State Machine*** (abbreviated CFSM), or sometimes a ***Characteristic Finite State Automata*** (abbreviated CFSA). The two terms are interchangeable.

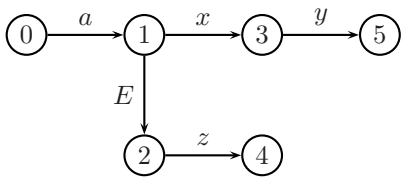
The machine is represented in the computer by a ***LR Parsing Table***. The rows of the table will be the states. The columns of the table will be the terminal and non-terminal symbols; these act as the transitions for our state machine. There will be three kinds of entries in the table.

- "S n" means to shift the token and go to state n.
- "R n" means to reduce by rule n.
- "n" by itself means to go to state n.
- A blank entry indicates an error.

Let's see how it would work for the grammar above. Here is the LR table corresponding to the grammar in section 2.

State	`a`	`x`	`y`	`z`	$	S	E
0	S 1						
1		S 3					2
2				S 4			
3			S 5				
4					R 1		
5				R 2			

The $ entry represents the end of the input. We can diagram the state machine.



To use this we need to keep track of our current state, the states we've been to before (since this is a push-down automata), the symbols we've read, and of course the input.

Using the same grammar as above, here is how the parse will run. We start off in state 0, an empty symbol stack, an empty state stack, and a full input queue.

State: 0
Input: `a x y z $`
State Stack:
Symbol Stack:

We look at the input and see the initial `a`. The LR table tells us (look at row 0, column `a`) to shift the input, then go to state 1.

State: 1
Input: `x y z $`
State Stack: 0
Symbol Stack: a

Now the parser looks at the input and sees the `x`. The LR table tells us (row 1, column `x`) to shift, and then go to state 3.

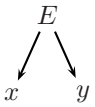
State:	3
Input:	`y z` $
State Stack:	0 1
Symbol Stack:	$a\ x$

Next, the parser shifts `y`, and moves to state 5.

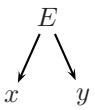
State:	5
Input:	`z` $
State Stack:	0 1 3
Stack:	$a\ x\ y$

At this point, the LR table has a reduce action. The "2" says to use rule 2 (i.e., $E \rightarrow x\ y$) for the reduction. There are two symbols on the RHS of this rule, so the parser will pop two symbols off the symbol stack, and also pop two states off the state stack.

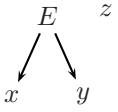
This causes the `x` and `y` to be reduced to an E, and the machine to back up to state 1.

State:	1
Input:	`z` $
State Stack:	0
Symbol Stack:	a 

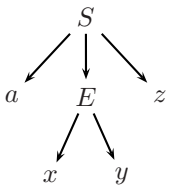
After a reduction occurs, the parser checks the LR table for the non-terminal symbol we just reduced. Row 1, column E has a 2 in it, so we go to state 2.

State:	2
Input:	`z` $
State Stack:	0 1
Symbol Stack:	a 

Next, the parser shifts `z`, and moves to state 4.

State:	4
Input:	$
State Stack:	0 1 2
Stack:	a  z

The entry in the table for this state says to reduce by rule 1. Since rule 1 ($S \rightarrow a\ E\ z$) has three symbols on the RHS, we pop three symbols from the symbol stack and combine them into the S node, and pop three states off the state stack. This gives up this

State:	0
Input:	$
State Stack:	
Symbol Stack:	

At this point, the parse is complete. Notice how the parser started with the leaves and built the branches from them. For this reason, LR parsing is also called ***bottom-up parsing***.

4 Generating the Characteristic Finite State Machine

This section explains how the table is actually generated, and the meaning of the different states.

4.1 The Algorithm

Each state of the CFSM represents a certain amount of progress in constructing a non-terminal. In order to generate the table (and thus the state machine) we need a way of representing what progress has been made in each state, and what remains to be done.

To do this, we will use a textual representation called an ***item***. An item is a rule from the grammar annotated with a "cursor" symbol •. The cursor represents where the parser is in building the non-terminal. For example, the item $S \rightarrow a\ E \bullet z$ shows that the parser is building an S, that an a and an E are on the stack, and that the next thing on the input is expected to be a z.

An ***initial item*** for a production $A \rightarrow \alpha$ is the item $A \rightarrow \bullet\ \alpha$. If the cursor is in front of any symbol, we will say that the item is ***advanceable***.

Each state in the machine will have one or more items associated with it. These will be called ***item sets***.

Here is the algorithm. It will create the item sets and fill in the rows of the parsing table.

1. Take all the productions for the start symbol, create initial items for them, take the transitive closure of the non-terminals[2], and put them into state 0.
2. For each advanceable item i in state s_i:
   (a) Create a new state s_n. Suppose the cursor in item i is in front of some symbol x.
   (b) If x is a terminal symbol:
   - Add the action "Shift s_n" to row s_i, column x of the parsing table.
   - For each item j in s_i where the cursor is in front of x, create a new item by advancing the cursor across x in item j. Add these new items to state s_n.

   (c) If x is a non-terminal symbol:
   - Add the number s_n to row s_i, column x in the parsing table. This is a "goto" action.
   - For each item j in s_i where the cursor is in front of x, create a new item by advancing the cursor across x in item j. Add these new items to state s_n.

   (d) Take the transitive closure of s_n.
   (e) If s_n is a duplicate of a state already in our set of states, discard it, and update the table accordingly.
3. If the state is not advanceable, then let X be the non-terminal the item is constructing, and let r be the corresponding rule from the grammar. Add the action "Reduce r" to row s_i, and to every column $c \in FOLLOW(X)$.
4. Repeat until all items have been considered.

Here are the item sets for the grammar we used in the parsing example above. It is customary to label state n as I_n.

$$
\begin{array}{llll}
I_0 & S & \rightarrow & \bullet\ a\ E\ z \\
I_1 & S & \rightarrow & a\ \bullet\ E\ z \\
 & E & \rightarrow & \bullet\ x\ y \\
I_2 & S & \rightarrow & a\ E\ \bullet\ z \\
I_3 & E & \rightarrow & x\ \bullet\ y \\
I_4 & S & \rightarrow & a\ E\ z\ \bullet \\
I_5 & E & \rightarrow & x\ y\ \bullet
\end{array}
$$

[2]We'll explain this shortly.

4.2 Examples

Here are some simple examples to illustrate the concepts.

4.2.1 A Grammar with a Single Rule

Consider the following simple grammar.

1. $A \rightarrow x\ y$.

During the parse, there will be three distinct stages.

To start, we create a table with a column for each of the symbols that appear on the RHS of our rules, together with \$, the end of input symbol.

Now we create our first state, making an initial item from the rule.

$$
\begin{array}{llll}
I_0 & A & \rightarrow & \bullet\ x\ y
\end{array}
$$

This state indicates that no input has been read.

Visiting State 0 Looking at the item in state 0, we advance the cursor across x and create state 1. We also put "shift 1" into the x column of row 0 of the parsing table.

Our item set now looks like:

$$
\begin{array}{llll}
I_0 & A & \rightarrow & \bullet\ x\ y \\
I_1 & A & \rightarrow & x\ \bullet\ y
\end{array}
$$

and our LR table now looks like:

State	x	y	\$
0	$S1$		
1			

Visiting State 1 Now we visit the item in state 1. We advance the cursor across y, creating state 2. We also put "shift 2" into the y column of row 1 of the parsing table.

Our item set now looks like:

$$
\begin{array}{llll}
I_0 & A & \rightarrow & \bullet\ x\ y \\
I_1 & A & \rightarrow & x\ \bullet\ y \\
I_2 & A & \rightarrow & x\ y\ \bullet
\end{array}
$$

and our LR table now looks like:

State	x	y	\$
0	$S1$		
1		$S2$	
2			

Visiting State 2 State 2 has a reduce operation, because the cursor is at the end of its item. The follow set of A is {$}, so we add "Reduce 1" (the "1" is the rule number) to the $ column of the second row. We now have the following parsing table.

State	x	y	$
0	$S1$		
1		$S2$	
2			$R1$

The state machine diagram corresponding to the CFSM looks like:

In practice we will not diagram these machines.

4.2.2 Two Productions for the Same Symbol

Consider the grammar

1. $A \rightarrow x\ y$
2. $A \rightarrow z$

Create State 0 We create state 0 by adding the initial items for the start symbol A.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y \\ & A & \rightarrow & \bullet\ z \end{array}$$

Visit State 0 The first item in state 0 creates state 1 when we shift the `x`. The second item in state 0 creates state 2 when we shift the `z`.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y \\ & A & \rightarrow & \bullet\ z \\ I_1 & A & \rightarrow & x\ \bullet\ y \\ I_2 & A & \rightarrow & z\ \bullet \end{array}$$

State	x	y	z	$
0	$S1$		$S2$	
1				
2				

Visit State 1 The item in state 1 creates state 3 when we shift the `y`.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y \\ & A & \rightarrow & \bullet\ z \\ I_1 & A & \rightarrow & x\ \bullet\ y \\ I_2 & A & \rightarrow & z\ \bullet \\ I_3 & A & \rightarrow & x\ y\ \bullet \end{array}$$

State	x	y	z	$
0	$S1$		$S2$	
1		$S3$		
2				
3				

Visit State 2 The item in state 2 is a reduction by rule 2. $Follow(A) = \{\$\}$, so we add "reduce 2" to the $ column of state 2.

State	x	y	z	$
0	$S1$		$S2$	
1		$S3$		
2				$R2$
3				

Visit State 3 The item in state 3 is a reduction by rule 1. $Follow(A) = \{\$\}$, so we add "reduce 1" to the $ column of state 3.

State	x	y	z	$
0	$S1$		$S2$	
1		$S3$		
2				$R2$
3				$R1$

We are out of items, so we are done.

You will notice that, except for the initial state, all the states are created from an item from a previous state. The order in which we visited the states is a canonical order. We started with the item at the top and visited each one in order. So, for example, we visited both of the items in state 0 before we visited the item in state 1. While the order doesn't matter as far as correctness is concerned, everyone who looks at your CFSM will expect it to be done in this order, and it will cause unnecessary confusion if you use a different order instead.

4.2.3 Solving the Common Prefix Problem

This grammar is not LL, because it has the common prefix problem. Here's how the LR table handles it.

1. $A \rightarrow x\ y$
2. $A \rightarrow q$
3. $A \rightarrow x\ z$

Create State 0 We create state 0 by adding the initial items of A to it.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y \\ & A & \rightarrow & \bullet\ q \\ & A & \rightarrow & \bullet\ x\ z \end{array}$$

State	x	y	z	q	\$
0					

Visit State 0 We shift the `x` for the first item in state 0. But, the third item also has a cursor in front of `x`. This means that we could be producing A according to the first rule, or we could be producing A according to the third rule. So, state 1 needs to have *both* of these items in it. Think about how the result is similar to the left-factoring technique we used for fixing common prefixes for LL parsers.

Also, the second item in state 0 creates state 2 when we shift `q`.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y \\ & A & \rightarrow & \bullet\ q \\ & A & \rightarrow & \bullet\ x\ z \\ I_1 & A & \rightarrow & x\ \bullet\ y \\ & A & \rightarrow & x\ \bullet\ z \\ I_2 & A & \rightarrow & q\ \bullet \end{array}$$

State	x	y	z	q	\$
0	$S1$			$S2$	
1					
2					

Visit State 1 Using the first item in state 1, we shift `y` and create state 3. Using the second item in state 1, we shift `z` and create state 4.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y \\ & A & \rightarrow & \bullet\ q \\ & A & \rightarrow & \bullet\ x\ z \\ I_1 & A & \rightarrow & x\ \bullet\ y \\ & A & \rightarrow & x\ \bullet\ z \\ I_2 & A & \rightarrow & q\ \bullet \\ I_3 & A & \rightarrow & x\ y\ \bullet \\ I_4 & A & \rightarrow & x\ z\ \bullet \end{array}$$

State	x	y	z	q	\$
0	$S1$			$S2$	
1		$S3$	$S4$		
2					
3					
4					

Visit State 2, 3, and 4 State 2 reduces rule 2. So we put "reduce 2" in the \$ column of state 2 (since {\$} is the follow set of A.) State 3 reduces rule 1. So we put "reduce 1" in the \$ column of state 3. State 4 reduces rule 3. So we put "reduce 3" in the \$ column of state 4.

State	x	y	z	q	\$
0	$S1$			$S2$	
1		$S3$	$S4$		
2					$R2$
3					$R1$
4					$R3$

At this point we are done.

4.2.4 Multiple Non-Terminals

In the grammars we have seen so far, we have used only terminal symbols in the RHS of the rules. When you have a non-terminal symbol, things are a little different. Here is an example.

Consider the grammar

1. $A \rightarrow x\ y\ A$
2. $A \rightarrow z$

Create State 0 We create state 0 by adding the initial items for the start symbol A.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y\ A \\ & A & \rightarrow & \bullet\ z \end{array}$$

State	x	y	z	\$	A
0					

Visit State 0 The first item creates state 1 when we shift the `x`. The second item creates state 2 when we shift the `z`.

$$\begin{array}{lllll} I_0 & A & \rightarrow & \bullet\ x\ y\ A \\ & A & \rightarrow & \bullet\ z \\ I_1 & A & \rightarrow & x\ \bullet\ y\ A \\ I_2 & A & \rightarrow & z\ \bullet \end{array}$$

State	x	y	z	\$	A
0	$S1$		$S2$		
1					
2					

Visit State 1 State 3 is created when we shift the `y` in the item of state 1. State 3 is complex. It represents that there are two things going on. On one hand, we are trying to create an A out of an `x`, a `y`, and another A. On the other hand, we are about to start creating that second A, and there are two ways that it could be done. So, in addition to the $A \rightarrow x\ y\ \bullet\ A$ item, we need also the two items where the cursor is at the beginning of A.

$$
\begin{array}{lllll}
I_0 & A & \rightarrow & \bullet\ x\ y\ A \\
 & A & \rightarrow & \bullet\ z \\
I_1 & A & \rightarrow & x\ \bullet\ y\ A \\
I_2 & A & \rightarrow & z\ \bullet \\
I_3 & A & \rightarrow & x\ y\ \bullet\ A \\
 & A & \rightarrow & \bullet\ x\ y\ A \\
 & A & \rightarrow & \bullet\ z
\end{array}
$$

State	x	y	z	\$	A
0	$S1$		$S2$		
1		$S3$			
2					
3					

Visiting State 2 State 2 reduces rule 2, so we put "reduce 2" in the \$ column of the table.

State	x	y	z	\$	A
0	$S1$		$S2$		
1		$S3$			
2				$R2$	
3					

Visiting State 3 In the first item of state 3, the cursor will move across the A. This transition occurs after the A on the RHS has just been reduced, and creates state 4.

In the second item, shifting $A \rightarrow \bullet\ x\ y\ A$ results in $A \rightarrow x\ \bullet\ y\ A$. This is exactly the same as state 1, so we will recycle state 1 rather than make a new state. Similarly, if shifting $A \rightarrow \bullet\ z$ in the third item results in an item that is just like state 2, so we recycle that one as well.

$$
\begin{array}{lllll}
I_0 & A & \rightarrow & \bullet\ x\ y\ A \\
 & A & \rightarrow & \bullet\ z \\
I_1 & A & \rightarrow & x\ \bullet\ y\ A \\
I_2 & A & \rightarrow & z\ \bullet \\
I_3 & A & \rightarrow & x\ y\ \bullet\ A \\
 & A & \rightarrow & \bullet\ x\ y\ A \\
 & A & \rightarrow & \bullet\ z \\
I_4 & A & \rightarrow & x\ y\ A\ \bullet
\end{array}
$$

State	x	y	z	\$	A
0	$S1$		$S2$		
1		$S3$			
2				$R2$	
3	$S1$		$S2$		4
4					

Visiting State 4 The item of state 4 reduces rule 1, so we add "reduce 1" to the \$ column of row 4.

State	x	y	z	\$	A
0	$S1$		$S2$		
1		$S3$			
2				$R2$	
3	$S1$		$S2$		4
4				$R1$	

We are out of items, so we are finished.

4.2.5 Transitive Closures

As you saw from the last example, when we have an item like $X \rightarrow a\ \bullet\ B\ c$, then you have to add the initial items for all the B rules to the state as well. It could happen, though, that the initial B items also have the cursor in front of a non-terminal. In that case, you add the initial items for that non-terminal as well. Every time you add an item to the state, you have to check if more items need to be added. Rather that saying "keep going until there's nothing more to do," we more formally call it "taking the transitive closure."

Here's an illustrative example.

1. $S \rightarrow x\ A \mid q$
2. $A \rightarrow B\ c$
3. $B \rightarrow d\ A \mid d$

Creating State 0 As before, we create state 0 by adding the initial items for the start symbol.

$$
\begin{array}{lllll}
I_0 & S & \rightarrow & \bullet\ x\ A \\
 & & \mid & \bullet\ q
\end{array}
$$

State	*c*	*d*	*q*	*x*	$	*A*	*B*	*S*
0								

Visiting State 0 State 1 is where the transitive closure takes place. We add $S \rightarrow x \bullet A$ to state 1 because we shift `x` in state 0. This causes us to have to add the initial A item, which in turn caused us to have to add the initial B item.

State 2 is derived by shifting `q` in the second item of state 0.

$$
\begin{array}{lllll}
I_0 & S & \rightarrow & \bullet\ x\ A \\
 & & | & \bullet\ q \\
I_1 & S & \rightarrow & x\ \bullet\ A \\
 & A & \rightarrow & \bullet\ B\ c \\
 & B & \rightarrow & \bullet\ d\ A \\
 & & | & \bullet\ d \\
I_2 & S & \rightarrow & q\ \bullet
\end{array}
$$

State	*c*	*d*	*q*	*x*	$	*A*	*B*	*S*
0				*S*1				
1								
2								

Visiting State 1 State 3 is derived by moving A in the first item of state 1. State 4 is derived by moving B in the second item of state 1. The third and fourth items of state 1 shift `d` to create state 5. This state also has a transitive closure.

$$
\begin{array}{lllll}
I_0 & S & \rightarrow & \bullet\ x\ A \\
 & & | & \bullet\ q \\
I_1 & S & \rightarrow & x\ \bullet\ A \\
 & A & \rightarrow & \bullet\ B\ c \\
 & B & \rightarrow & \bullet\ d\ A \\
 & & | & \bullet\ d \\
I_2 & S & \rightarrow & q\ \bullet \\
I_3 & S & \rightarrow & x\ A\ \bullet \\
I_4 & A & \rightarrow & B\ \bullet\ c \\
I_5 & B & \rightarrow & d\ \bullet\ A \\
 & & | & d\ \bullet \\
 & A & \rightarrow & \bullet\ B\ c \\
 & B & \rightarrow & \bullet\ d\ A \\
 & & | & \bullet\ d
\end{array}
$$

State	*c*	*d*	*q*	*x*	$	*A*	*B*	*S*
0			*S*2	*S*1				
1		*S*5				3	4	
2								
3								
4								
5								

Visiting States 2 and 3 These states also have reduction items. State 2 reduces rule 2. The follow set of S is {$}. State 3 reduces rule 1.

State	*c*	*d*	*q*	*x*	$	*A*	*B*	*S*
0			*S*2	*S*1				
1		*S*5				3	4	
2					*R*2			
3					*R*1			
4								
5								

Visiting State 4 State 4 shifts `c` to make state 6.

$$
\begin{array}{lllll}
I_0 & S & \rightarrow & \bullet\ x\ A \\
 & & | & \bullet\ q \\
I_1 & S & \rightarrow & x\ \bullet\ A \\
 & A & \rightarrow & \bullet\ B\ c \\
 & B & \rightarrow & \bullet\ d\ A \\
 & & | & \bullet\ d \\
I_2 & S & \rightarrow & q\ \bullet \\
I_3 & S & \rightarrow & x\ A\ \bullet \\
I_4 & A & \rightarrow & B\ \bullet\ c \\
I_5 & B & \rightarrow & d\ \bullet\ A \\
 & & | & d\ \bullet \\
 & A & \rightarrow & \bullet\ B\ c \\
 & B & \rightarrow & \bullet\ d\ A \\
 & & | & \bullet\ d \\
I_6 & A & \rightarrow & B\ c\ \bullet
\end{array}
$$

State	*c*	*d*	*q*	*x*	$	*A*	*B*	*S*
0			*S*2	*S*1				
1		*S*5				3	4	
2					*R*2			
3					*R*1			
4	*S*6							
5								
6								

Visiting State 5 State 5 recycles several entries, including itself. The forth item in it, $B \rightarrow \bullet\ d\ A$, when visited, generates state 5 all over again.

The A is moved to make state 7, a B is moved to go to state 4, and `d` is shifted to to go back to state 5.

Further, rule 5 is reduced. The follow set of B is $\{c\}$.

$$
\begin{array}{lllll}
I_0 & S & \rightarrow & \bullet\ x\ A \\
 & & | & \bullet\ q \\
I_1 & S & \rightarrow & x\ \bullet\ A \\
 & A & \rightarrow & \bullet\ B\ c \\
 & B & \rightarrow & \bullet\ d\ A \\
 & & | & \bullet\ d \\
I_2 & S & \rightarrow & q\ \bullet \\
I_3 & S & \rightarrow & x\ A\ \bullet \\
I_4 & A & \rightarrow & B\ \bullet\ c \\
I_5 & B & \rightarrow & d\ \bullet\ A \\
 & & | & d\ \bullet \\
 & A & \rightarrow & \bullet\ B\ c \\
 & B & \rightarrow & \bullet\ d\ A \\
 & & | & \bullet\ d \\
I_6 & A & \rightarrow & B\ c\ \bullet \\
I_7 & B & \rightarrow & d\ A\ \bullet
\end{array}
$$

State	*c*	*d*	*q*	*x*	$	*A*	*B*	*S*
0			*S*2	*S*1				
1		*S*5				3	4	
2					*R*2			
3					*R*1			
4	*S*6							
5	*R*5	*S*5				7	4	
6								
7								

Visiting State 6 and 7 These are reductions. State 6 reduces rule 3. The follow set of A is {$,c}, so we put a reduction in both of those. State 7 reduces rule 4.

So, our final CFSM is as above, and our final LR Parsing table follows.

State	*c*	*d*	*q*	*x*	$	*A*	*B*	*S*
0			*S*2	*S*1				
1		*S*5				3	4	
2					*R*2			
3					*R*1			
4	*S*6							
5	*R*5	*S*5				7	4	
6	*R*3				*R*3			
7	*R*4							

5 Ambiguities

If the grammar is ambiguous, we will detect it during the generation of the LR table, because our algorithm will want to put two actions in the same location. If the algorithm tries to put a shift action and a reduce action in the same spot, then the result is a ***shift-reduce conflict***. These are fairly common. Much more rare is to have two reduce actions in the same spot. These are called ***reduce-reduce conflicts***.

Here is a grammar for expression that has a common ambiguity.

$$
\begin{array}{llll}
1. & S & \rightarrow & E\ ; \\
2. & E & \rightarrow & E\ +\ E \\
3. & E & | & E\ *\ E \\
4. & E & | & v
\end{array}
$$

See if you can create the CFSM and the LR Parsing tables. Note that the follow set of E is $\{+, *, ;\}$, and the follow set of S is {$}.

Here are the CFSM and LR Parsing Table.

$$
\begin{array}{lrcl}
I_0 & S & \rightarrow & \bullet\ E\ ; \\
 & E & \rightarrow & \bullet\ E\ +\ E \\
 & E & | & \bullet\ E\ *\ E \\
 & E & | & \bullet\ v \\
I_1 & S & \rightarrow & E\ \bullet\ ; \\
 & E & \rightarrow & E\ \bullet\ +\ E \\
 & E & | & E\ \bullet\ *\ E \\
I_2 & E & \rightarrow & v\ \bullet \\
I_3 & S & \rightarrow & E\ ;\ \bullet \\
I_4 & E & \rightarrow & E\ +\ \bullet\ E \\
 & E & \rightarrow & \bullet\ E\ +\ E \\
 & E & | & \bullet\ E\ *\ E \\
 & E & | & \bullet\ v \\
I_5 & E & \rightarrow & E\ *\ \bullet\ E \\
 & E & \rightarrow & \bullet\ E\ +\ E \\
 & E & | & \bullet\ E\ *\ E \\
 & E & | & \bullet\ v \\
I_6 & E & \rightarrow & E\ +\ E\ \bullet \\
 & E & \rightarrow & E\ \bullet\ +\ E \\
 & E & | & E\ \bullet\ *\ E \\
I_7 & E & \rightarrow & E\ *\ E\ \bullet \\
 & E & \rightarrow & E\ \bullet\ +\ E \\
 & E & | & E\ \bullet\ *\ E
\end{array}
$$

State	;	+	*	*v*	$	*E*	*S*
0				*S*2		1	
1	*S*3	*S*4	*S*5				
2	*R*4	*R*4	*R*4				
3					*R*1		
4				*S*2		6	
5				*S*2		7	
6	*R*2	*S*4/*R*2	*S*5/*R*2				
7	*R*3	*S*4/*R*3	*S*5/*R*3				

The trouble happens in states 6 and 7. The algorithm depends on the follow set of a symbol to determine if it is ready to reduce. But, the same symbol appears inside the rule as well.

From a higher level point of view, the problem occurs because we do not have precedence information about `+` and `*`. In modern parser generators, you can annotate the grammar to indicate which symbol should have precedence. If we chose the standard precedence, then when the parser has a choice between shifting `*` and reducing `+`, it will chose to shift.

If there is no indication given at all, then parser generators will shift, because in practice it is more likely to give correct results.